

APPLICATION AS FILED

108290-57096860

PATENT APPLICATION

**INLINE DATABASE FOR RECEIVER TYPES IN OBJECT-
ORIENTED SYSTEMS**

Inventors: **LARS BAK**
3782 Corina Way
Palo Alto, California 94303
a Citizen of Denmark

URS HÖLZLE
7220 Davenport Road, #105
Goleta, California 93117
a Citizen of Switzerland

Assignee: **SUN MICROSYSTEMS, INC.**
2250 Garcia Avenue
Mountain View, California 94043
a Corporation of the state of Delaware

Status: Large Entity

HICKMAN BEYER AND WEAVER, LLP
P.O. Box 61059
Palo Alto, CA 94306
Telephone (650) 493-6400

09856073.062801

PATENT

5

INLINE DATABASE FOR RECEIVER TYPES IN OBJECT-ORIENTED
SYSTEMS

CROSS REFERENCE TO RELATED APPLICATIONS

- 10 This application is related to U.S. Patent Application No. _____ (Atty. Docket No. SUN1P156/P2306), entitled "Site Specific Message Dispatch in Object-Oriented Systems," U.S. Patent Application No. _____ (Atty Docket No. SUN1P147/P2297), entitled "Method and Apparatus for Performing Byte-Code Optimization During Pauses," U.S. Patent Application No. _____ (Atty. Docket No. SUN1P150/P2300), entitled
- 15 "Mixed Execution Stack and Exception Handling," U.S. Patent Application No. _____ (Atty. Docket No. SUN1P152/P2302), entitled "Method and Apparatus for Implementing Multiple Return Sites," U.S. Patent Application No. _____ (Atty. Docket No. SUN1P168/P2840), entitled "Method and Apparatus for Dynamically Optimizing Byte-Coded Programs," U.S. Patent Application No. _____ (Atty. Docket No.
- 20 SUN1P175/P2841), entitled "Method and Apparatus for Dynamically Optimizing Compiled Activations," all filed concurrently herewith, U.S. Patent Application No. _____ (Atty. Docket No. SUN1P144/P2260), entitled "Interpreting Functions Utilizing a Hybrid of Virtual and Native Machine Instructions," filed June 30, 1997, and U.S. Patent Application No. _____ (Atty. Docket No. SUN1P151/P2301), entitled
- 25 "Interpreter Generation and Implementation Utilizing Interpreter States and Register Caching," filed June 30, 1997, which are all incorporated herein by reference for all purposes in their entirety.

BACKGROUND OF THE INVENTION

The present invention relates to the increasing the execution speed of object-oriented programs. More specifically, the invention relates to utilizing information collected during execution of an object-oriented program for subsequent executions.

5 The fundamental idea behind object-oriented languages is the combination of both data and the methods (or functions) that operate on that data into a single unit, which is called an object. An object's functions typically provide the only way to access the data that is encapsulated by the object. The data is accessed by sending a message to the object instructing the object to invoke the method specified by the message.

10 Efficient message dispatch is of paramount importance in object-oriented languages. This is because message dispatch is a very frequent operation in object-oriented programs and is performed at runtime; therefore, it should be as fast as possible. Message dispatch, however, is far from being a trivial operation. Unlike procedural programming languages (e.g., the C programming language) that can determine a
15 function's address before runtime, object-oriented languages must determine the method that handles a message that has been dispatched to a receiver object at runtime, and it may involve an extensive search.

In order to better understand the complexities of message dispatch, Fig. 1 shows a class hierarchy including methods of each class. A class hierarchy 1 includes at its root a
20 parent class A 3 that defines two virtual functions foo() and bar(). Virtual functions are functions that may be defined in a parent class and redefined in the children classes. Classes B 5 and class C 7 inherit the data and methods of the parent class A. As shown, class B does not redefine either of the virtual functions foo and bar. However, class C redefines the virtual function foo. When an object of class C is requested to invoke the
25 method foo, the method invoked will be the method defined by the class C, not the parent class A. Classes D 9 and E 11 also redefine the method foo.

As it is generally impossible to determine the class of an object statically, the search for the correct method is performed at runtime, during message dispatch. There are

many known techniques for implementing method dispatch. For example, Fig. 2 shows an inline cache. Assume a method 51 was originally as follows:

```
main()
{
...
x.foo();
...
}
```

Thus, the method main includes a statement `x.foo()` in order to invoke the method `foo` of object `x`.

During runtime, the system would have to determine to which class object `x` belongs before the method to handle the method could be invoked. With an inline cache, the first time the system determines the class to which object `x` belongs, a direct call to the class's method is written into the computer code.

Assuming that object `x` is a member of class `A`, the call `x.foo()` is changed to a direct call `A::foo()`. The arrow specifies a method `foo` for class `A`. Since the object `x` may not be of class `A` every time, a prolog verifies that the object `x` is of the correct class, which is represented as the expression `x=A` meaning that it is determined if the class of object `x` is equal to class `A`. The class of an object may be determined from a value stored in the object. If the object is of the correct class, a jump is performed to method code 57 that handles the message.

Returning to prolog 55, if the object is not of class `A`, a method lookup routine is called in order to determine the correct method. Once the correct method is found, the system updates the message dispatch (or call) site with a direct call to that method. Additionally, the system updates the prolog to specify the new class. As an example, assume that the first time that the system encountered `x.foo()`, object `x` was of class `A` and the data structures were modified as shown in Fig. 2.

Once the data structures are modified as shown, subsequent calls to `x.foo()` will be substantially more efficient if object `x` is of class `A`. However, if object `x` subsequently is of class `B`, prolog 55 calls a method lookup routine to find the method and let's assume it determines that object `x` is now of class `B`. Referring again to Fig. 1, it is seen that the

method foo for class B is the same method foo as defined in class A (*i.e.*, class B did not redefine the virtual function foo). Accordingly, the message dispatch in method 51 will be changed to B::foo () and the condition in prolog 55 will be changed to x=B.

An inline cache may be an efficient way of implementing message dispatch if the object at a call site bar remains the same class. However, if the object is of multiple classes, the system is continually calling the method lookup routine and patching the call site and prolog. Thus, the system may be actually less efficient.

Another technique for implementing message dispatch is the use a polymorphic inline cache as shown in Fig. 3. As before, a method 101 originally included a method dispatch x.foo(). With a polymorphic inline cache, a stub 103 is generated that is able to perform the message dispatch for different receiver types. The original message dispatch is overwritten with a call to the polymorphic inline cache stub 103. Each time a new receiver type is encountered, a statement is added to the stub. As shown, three different receiver types of have been encountered thus far. If the receiver type and has been encountered, a call is made to the method to handle the message for that receiver type. Otherwise, the method lookup routine is called to determine the appropriate method to handle the message. Typically, a new statement will be added to stub 103 in order to handle each new receiver type.

The polymorphic inline cache is more flexible than the inline cache as it is able to handle multiple receiver types. However, a drawback of the polymorphic inline cache is that as more receiver types are encountered, the stub continues to grow and it becomes less and less efficient at performing message dispatch. For example, the system may need to plow through multiple if statements before finding the right method to handle the message.

Fig. 4 shows another message dispatch technique called hashing. In hashing, the original message dispatch in a method 151, x.foo(), is overwritten with a call to a hash function 153. The hash function hashes the receiver type and the message in order to form a hash key, which is typically an index into a hash table 155. The hash table includes an index 157, receiver types 159, messages 161, and methods 163. Once the hash function

hashes to a row in cash table 155, the receiver type and message are retrieved from columns 159 and 161 of the hash table. If the receiver type and message at the call site match the receiver type and message in the row of the hash table, the method specified in column 163 of hash table 155 is invoked. Otherwise, a method lookup routine is called in order to find the correct method. Typically, the new method is then added to the hash table.

Although hashing is the most flexible message dispatch technique we have described thus far, it is more computationally and storage intensive than the other techniques. Another drawback of the message dispatch techniques we have described is that none of the techniques are site specific. In other words, none of the message dispatch techniques provide the flexibility of handling message dispatches at different call sites in a different manner. Other method dispatch techniques are described in "Message Dispatch on Pipelined Processors," by K. Driesen et al., ECOOP, 1995, which is hereby incorporated by reference for all purposes.

None of the message dispatch techniques described above allow for the receiver type information that is collected to be utilized during a subsequent execution of the program. Additionally, it would be desirable if inlining information was available during subsequent program executions. Accordingly, there is a need for techniques that allow for collected receiver type and/or inlining information to be utilized in later runs of the program.

SUMMARY OF THE INVENTION

Embodiments of the present invention provide innovative techniques for utilizing collected receiver type and/or inlining information during later program executions. By allowing a program to take advantage of information from previous executions, the program may become quickly optimized without requiring the information to be collected again. Therefore, an object oriented-program may become increasing more efficient each time it is run until it reaches a steady optimized state. Several embodiments of the invention are described below.

In one embodiment, the invention provides a method of implementing message dispatch for an object-oriented program. Receiver type information is collected at a site of a method that dispatches message to receiver objects. The collected receiver type information is then saved for a subsequent execution of the program. In preferred embodiments, the receiver type information is collected both during interpreted and compiled execution of the method.

In another embodiment, the invention provides a method of implementing message dispatch for an object-oriented program. During interpretation, collecting receiver type information at a site of a method that dispatches messages to receiver objects. The receiver type information may include each different receiver type and a reference to the site for each different receiver type to which messages were dispatched from the site. It is then determined that it would be desirable to compile the method that includes the site that dispatches message to receiver objects. When the method is compiled, the receiver type information is included at the site in the compiled method. Also, the receiver type information is saved for a subsequent execution of the program. In some embodiments, the receiver type information is collected in a polymorphic inline cache.

In another embodiment, the invention provides a data structure stored by a computer readable medium. At least one receiver type is stored by the computer readable medium with nested receiver types. The nested receiver types are of receiver types that

were dispatched messages at message dispatch sites in a method. In preferred embodiments, the data structure is saved in an attributes section of a Java class file.

Other features and advantages of the invention will become readily apparent upon review of the following detailed description in association with the accompanying

5 drawings.

09396075-062801

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates a class hierarchy of classes including virtual functions in an object-oriented environment.

5 Fig. 2 illustrates an inline cache technique for object-oriented message dispatch.

Fig. 3 illustrates an a polymorphic inline cache technique for object-oriented message dispatch.

Fig. 4 illustrates a hashing technique for object-oriented message dispatch.

10 Fig. 5 illustrates an example of a computer system that may be utilized to execute the software of an embodiment of the invention.

Fig. 6 illustrates a system block diagram of the computer system of Fig. 5.

Fig. 7 illustrates how a Java source code program is executed.

Fig. 8 illustrates an embodiment of a site specific inline cache technique for object-oriented message dispatch.

15 Fig. 9 shows a high level flowchart of an embodiment of a site specific inline cache technique for object-oriented message dispatch.

Fig. 10 shows a high level flowchart of an embodiment of a site specific message dispatch technique.

20 Fig. 11 shows a flowchart of dynamically determining if another message dispatch technique is desirable at a specific site.

Fig. 12 illustrates an embodiment of a site specific message dispatch technique that utilizes NOP instructions to count the number of different receiver types that have been encountered.

25 Fig. 13 shows a high level flowchart of an inline database that may be utilized to save receiver type information for subsequent execution of a program.

Fig. 14 shows a flowchart of collecting receiver type information during interpretation and compiled execution of a program for an inline database.

Fig. 15 illustrates a method that has been compiled to include a polymorphic inline cache technique for object-oriented message dispatch.

Fig. 16 illustrates a Java class file in which receiver type information has been stored in the attributes section of the class file.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Definitions

5 Function – A software routine (also called a subroutine, procedure, member function, and method).

 Message dispatch – A process of determining the method to handle a message that has been sent to the object into an object-oriented environment.

 Receiver object (or receiver) – The object that has been sent a message in an
10 object-oriented environment.

 Receiver type – The class to which a receiver object belongs.

 Call site (or site) – The location or area in a program where a message is dispatched to an object.

15 Overview

 In the description that follows, the present invention will be described in reference to preferred embodiments that are designed for a Java™ virtual machine. In particular, examples will be described that are directed to IBM personal computers. However, the invention is not limited to any particular language, computer architecture, or specific
20 implementation. Therefore, the description of the embodiments that follows for purposes of illustration and not limitation.

 Fig. 5 illustrates an example of a computer system that may be used to execute the software of an embodiment of the invention. Fig. 5 shows a computer system 301 that includes a display 303, screen 305, cabinet 307, keyboard 309, and mouse 311. Mouse
25 311 may have one or more buttons for interacting with a graphical user interface. Cabinet 307 houses a CD-ROM drive 313, system memory and a hard drive (*see* Fig. 6) which may be utilized to store and retrieve software programs incorporating computer code that implements the invention, data for use with the invention, and the like. Although CD-ROM 315 is shown as an exemplary computer readable storage medium, other computer

readable storage media including floppy disk, tape, flash memory, system memory, and hard drive may be utilized. Additionally, a data signal embodied in a carrier wave (e.g., in a network including the Internet) may be the computer readable storage medium.

Fig. 6 shows a system block diagram of computer system 301 used to execute the software of an embodiment of the invention. As in Fig. 5, computer system 301 includes monitor 303 and keyboard 309, and mouse 311. Computer system 301 further includes subsystems such as a central processor 351, system memory 353, fixed storage 355 (e.g., hard drive), removable storage 57 (e.g., CD-ROM drive), display adapter 359, sound card 361, speakers 363, and network interface 365. Other computer systems suitable for use with the invention may include additional or fewer subsystems. For example, another computer system could include more than one processor 351 (i.e., a multi-processor system) or a cache memory.

The system bus architecture of computer system 301 is represented by arrows 367. However, these arrows are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be utilized to connect the central processor to the system memory and display adapter. Computer system 301 shown in Fig. 6 is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

The Java™ programming language was developed by Sun Microsystems. Typically, computer programs written in the Java programming language are compiled into bytecodes or Java virtual machine instructions that are then executed by a Java virtual machine. The bytecodes are stored in class files that are input into the Java virtual machine for interpretation. Fig. 7 shows a progression of a simple piece of Java source code through execution by an interpreter, the Java virtual machine.

Java source code 401 includes the classic Hello World program written in Java. The source code is then input into a bytecode compiler 403 that compiles the source code into bytecodes. The bytecodes are virtual machine instructions as they will be executed by a software emulated computer. Typically, virtual machine instructions are generic (i.e., not designed for any specific microprocessor or computer architecture) but this is not

required. The bytecode compiler outputs a Java class file 405 that includes the bytecodes for the Java program.

The Java class file is input into a Java virtual machine 407. The Java virtual machine is an interpreter that decodes and executes the bytecodes in the Java class file.

- 5 The Java virtual machine is an interpreter, but is commonly referred to as a virtual machine as it emulates a microprocessor or computer architecture in software (e.g., the microprocessor or computer architecture may not exist in hardware). In order to increase execution speed of the Java virtual machine, methods may be compiled into native machine instructions for faster execution. In the description of preferred embodiments
10 that follows, compilation will be used to refer to the translation of Java virtual machine instructions into native machine instructions.

Site Specific Inline Cache

- The invention provides a site specific inline cache technique for implementing
15 message dispatch in object-oriented environments. Receiver type information is stored at the call site, which allows each call site to vary according to runtime characteristics of the call site. This allows message dispatch to not only be more efficient, but also to be more flexible.

- Fig. 8 illustrates a site specific inline cash technique according to one embodiment
20 of the invention. As described in the above in the Background of the Invention, a method 51 originally included a message x.foo() for dispatch. The first time that this message is dispatched, the system may call a method lookup routine to determine the location of the appropriate method to handle the message. Alternatively, the system may perform a static analysis prior to runtime to determine what is likely to be the correct method.

- 25 With a site specific inline cache, once the class of the object that should handle the message is determined, an instruction is generated at the call site to save the receiver type or class. The message dispatch is then overwritten with a call to the specific method that should handle the message. Thus, a message dispatch becomes two instructions, one that saves a receiver type and a call to the appropriate method.

As shown, method 451 includes a move instruction that stores an id for the class A into a special register. The register is called a special register only for identification purposes. What is important is that if a register is utilized, the register should be otherwise unused at this call site. Although utilizing registers may provide the most efficient execution speed, other storage mechanisms may be utilized. For example, the receiver type may be stored in system memory or sent as a parameter to the message handling method.

Once the receiver type is saved, a call is made to a method 453 that handles the message, which is shown as A::foo(). A prolog 455 checks if the object x is of the saved receiver type, which is retrieved from the special register. In the example shown class A is the stored receiver type. Thus, if object x is of class A, a jump is made to a method code 457 of A::foo(). Otherwise, a method lookup routine is called to locate the appropriate method to handle the message. Then, the call site may be updated to save the receiver type of object x and a call to the method found by the method lookup routine.

The receiver type stored at the call site is therefore a predicted receiver type. It is the receiver type that the system predicts will be the receiver type of the next object to receive the message. If the predicted receiver type matches the receiver type of the object, the message dispatch is more efficient. Additionally, each message dispatch site may have receiver type information stored that is particular to that call site and another site will not overwrite the stored receiver type information.

Fig. 9 shows a flowchart of an embodiment of the site specific inline cache technique for message dispatch. At a step 501, a predicted receiver type is saved at a site where a message is dispatched to a receiver object. In preferred embodiments, the receiver type is saved by utilizing a move instruction to place the receiver type in a register. However, other method of saving the receiver type may be utilized that are well known to those of skill in the art, including the use of system memory or parameter passing.

At a step 503, the message is sent to the receiver object. The message is sent by a direct call to (or invocation of) the method that handles the message instead of utilizing a method lookup routine.

Once the receiver object receives the message, it is verified that the receiver object is of the saved predicted receiver type at a step 505. In preferred embodiments, this verification is performed in the prolog of the method (see Fig. 8). If the receiver object is of the same type as the saved predicted receiver type at a step 507, the method that handles the message is executed at a step 509. Step 509 may include performing a jump to the start of the method code that handles the message.

If the receiver object is not of the same type as the saved predicted receiver type, the predicted receiver type at the call site is changed to the receiver type of the receiver object at a step 511. The method lookup routine is called to obtain a new method at a step 513. The method lookup routine utilized may be any that are known to those of skill in the art, including dispatch table searches and virtual function tables. The method lookup routine provides the appropriate method to handle the message. The message dispatch at the call site is then changed to the method returned by the method lookup routine at a step 515.

As an example, assume that in Fig. 8, object x is of class A so the predicted receiver type stored at the call site in method 451 is class A. However, if object x is subsequently of class B, prolog 455 may modify the move instruction to store the class B in the special register as the predicted receiver type. Additionally, the message dispatch will be modified to invoke the method returned by the method lookup routine. Subsequent message dispatches at this call site that are directed at a receiver object of class B will then be more efficiently dispatched.

In Fig. 9, the method is shown as being executed after step 515 that changes the method at the site. The step of executing the method may actually occur within step 513 or at other locations. Therefore, in the flowcharts shown, steps may be reordered, added and deleted according to the specific implementation without departing from the spirit of the invention.

As described above, a site specific inline cache technique is provided that saves a predicted receiver type at a call site. By storing a predicted receiver type at a call site, message dispatch may be more efficient and flexible. For example, the operations at each call site may be insulated from each other so that one call site will not change the settings for another call site.

Site Specific Message Dispatch

The invention provides a site specific message dispatch technique that is flexible in allowing for different message dispatch techniques to be utilized at each site. Additionally, the message dispatch technique utilized at a call site may change over time, during runtime. Once it is determined that a new or different message dispatch technique is desirable at a specific site, the new message dispatch technique is set up for use at the specific call site. Thus, message dispatch becomes more flexible not only in terms of call site, but also in terms of the message dispatch technique that is utilized at call during different times during program execution.

Fig. 10 shows a high level flowchart of an embodiment of site specific message dispatch. At a step 511, a first message dispatch technique is performed at a specific message dispatch site. The first message dispatch technique may be any number of message dispatch techniques known in the art. Preferably, the first message dispatch technique is a technique according to the present invention. For example, the first message technique may be an embodiment of the site specific inline cache described above. The first message dispatch technique may be determine statically before runtime or dynamically during runtime.

At a step 553, the system dynamically during runtime determines if a second message dispatch technique is desirable at the specific site. Any number ways may be utilized to determine that a second message dispatch technique is desirable. For example, once the number of different receiver types encountered at this call site crosses a threshold, it may be desirable to switch to a different message dispatch technique. Additionally, a calculation may be performed on the frequency that the method lookup

routine is called so that if the frequency is too high, a new message dispatch may be warranted.

At any rate, once it is determined that a second message dispatch technique is desirable, the second message dispatch technique is performed at the specific site at a step

5 555. The mechanism for switching to a new message dispatch technique typically depends on the new technique itself. Now that the high level flow has been described, it may be beneficial to discuss in more detail an embodiment of the step of dynamically determining if a second message dispatch technique is desirable.

Fig. 11 shows an embodiment of dynamically determining if a second message
10 dispatch technique is desirable. The labels of first and second message dispatch
techniques are presented as generic indications that one message dispatch technique may
be switched to another at a specific call site. However, the invention is not limited to only
two techniques, and may be advantageously applied to three or more techniques.
Therefore, the labels of first and second are not an indication that the invention is limited
15 to only two message dispatch techniques.

The embodiment shown in Fig. 11 relies on the number of different receiver types encountered at a specific call site as an indicator of when the message dispatch should be changed. As an example, a site specific inline cache technique may be utilized until five different receiver types have been encountered at this call site. When different receiver type is utilized in this context, it means that there have been five instances where the predicted receiver type has been different than the actual receiver type. This would mean that approximately five times, the method lookup routine has been called and the predicted receiver type that is stored at the call site has changed. As an example, a call site that alternates five times between receiver objects of classes A and B may be counted as five different receiver types.

Although a specific receiver type counter may be utilized at each call site, this may entail the use of memory and computer code to maintain the counter. In preferred embodiments, the counter is implemented as multiple no-operation (NOP) instructions preceding the method that handles the message. As will be described in more detail in

reference to Fig. 12, the number NOP instructions that implemented before the method indicates the number of different receiver types that have been encountered.

Referring to Fig. 11, the flowchart assumes that the method lookup routine has been called, otherwise, most likely, the current message dispatch technique is performing efficiently. At a step 601, the system dispatches messages to NOP instructions preceding the method that handles the message. The system counts the number of different receiver types by determining how many NOP instructions are executed before the method. Typically, this count will be determined by subtracting the address to where the message was dispatched for this site from the address of the actual method, including any prolog.

At a step 605, the count is compared to a predetermined number. The predetermined number is the number of different receiver types that may be encountered at a call site before it is desirable to switch to a different message dispatch technique. The predetermined number may be statically defined or calculated dynamically during runtime.

If the count is greater than the predetermined number at a step 607, a switch is made to the second message dispatch technique at a step 609. If the count is less than or equal to the predetermined number, this indicates that, at present, it is not desirable to switch to the second message dispatch technique. An example may help in illustrating how NOP instructions may be utilized as a counter.

Fig. 12 illustrates how NOP instructions may be utilized as a counter of the number of different receiver types that have been encountered at a specific call site. A method 651 is shown with a site specific inline cache as described previously. A method 653 handles the message, however, the method is preceded by a NOP section 655 that has multiple NOP instructions. NOP instructions may be chosen as they do nothing and may not unduly effect the operating speed of the method. Other instructions may be utilized instead of NOP instructions in other embodiments.

NOP section 655 precedes a prolog 657 and method code 659 of method 653. The NOP section is composed of a predetermined number of NOP instructions, typically the same number as the number of different receiver types that may be encountered at a call

09395075-062801

site before a switch to a different message dispatch technique is initiated. Each time the method lookup routine is called in prolog 657, the move instruction is patched with the receiver type of object x. Additionally, the subsequent message dispatch (e.g., A::foo()) is changed to one of the NOP instructions in NOP section 655 of the new method found by the method lookup routine.

As shown, the message dispatch points to the third NOP instruction before method 653. If prolog 657 determines that object x is not of class A, the prolog calculates the difference between the address specific at the call site and the beginning of method 653, which is obtainable during runtime by a number of known methods. The difference or count indicates the number of different receiver types that have been encountered at this specific call site. If the count is greater than a predetermined number, prolog 657 may switch the message dispatch technique utilized at this call site.

Utilizing NOP instructions before a method is easy to implement, fast, and does not require an excessive amount of computer code to keep the counter. Additionally, NOP instructions in NOP section 655 may be referenced by multiple different call sites without any impact on each other.

The first message dispatch technique was shown as being a site specific inline cache. The second message dispatch technique may be a polymorphic inline cache or hashing technique, as each are designed to handle multiple receiver types. Of course, there is no limitation on the message dispatch techniques that may be utilized with the invention. The specific message dispatch techniques have been described herein to aid the reader's understanding.

As described above, a site specific message dispatch technique is provided that allows for a new message dispatch technique to be utilized at a specific call site when it is deemed desirable. By allowing each call site to switch message dispatch techniques when it becomes desirable, message dispatch may be more efficient and flexible. Additionally, the operations at each call site may be insulated from each other so that one call site will not change the settings for another call site.

Inline Database

The invention provides a technique of saving receiver type and/or inlining information for a subsequent execution of a program. Receiver type and inlining information may be collected during the runtime of the program, the collected runtime
5 information may then be stored so that a subsequent execution of the program can realize the benefits of the collected information without having to collect the information again. This allows the program to achieve an optimized state during runtime and quickly return to that optimized state in a subsequent execution of the program.

Fig. 13 shows a high level flowchart of an embodiment of the invention that
10 collects receiver type information and stores it for a subsequent execution of a program. At a step 701, the system collects receiver type information at a message dispatch site of a method of a program. The receiver type information may be collected by any of the message dispatch techniques described here or otherwise known in the art. The receiver type information is preferably specific to a call site so receiver type information may be
15 collected for multiple call sites.

Once the receiver type information for a specific call site is collected, it is saved for a subsequent execution of the program at a step 703. The receiver type information may be saved at the normal termination of the program or at some point during runtime, either with or without a user's input. It may be beneficial to describe in more detail an
20 embodiment of collecting receiver type information.

Fig. 14 shows a flowchart of an embodiment of collecting receiver type information. At a step 751, receiver type information is collected during interpretation of the method. Typically, methods are interpreted in a Java virtual machine until it is determined that it would be beneficial to compile the method. In systems where methods
25 are immediately compiled, the flow may begin at a subsequent step.

As receiver type information is collected during interpretation, such as the different receiver types that were encountered at specific call sites, one or more than one message dispatch technique may be utilized. At some point, there may be a question of whether this method should be compiled at a step 753. For example, if a method has been

interpreted more than a predetermined number of times, it may be determined that the method should be compiled. Otherwise, receiver type information may be continually collected during interpretation of the method.

At a step 755, the method is compiled and the collected receiver type information is compiled into the method. This way, the compiled method will realize the benefits of the collected type information. Additionally, in preferred embodiments, computer code is generated in the compiled method to continue collecting type information at a step 757.

In preferred embodiments, the system also maintains a working set of compiled methods. This working set may also be saved off with the collected type information so that in a subsequent execution of the program, the system may compile methods from the working set when the system has available processing capabilities (*e.g.*, when the system is waiting on user input).

Once a method is compiled, it may be beneficial to recompile the method at some point in time. At a step 759, it is determined if the method should be recompiled. As an example, it may be beneficial to recompile the method if a substantial amount of new receiver type information has been collected. In preferred embodiments, a method may be recompiled if it is determined (*e.g.*, heuristics) that a method should be inlined into this method. Inlining means that instead of calling the method, the method is compiled into the calling method. For example, if it is determined that it would be beneficial to inline a method, the method is compiled into the calling method and thus the calling method now includes the call sites of the inlined method.

It is difficult, if no impossible for conventional object-oriented systems to inline methods since the receiver object may not be determined till runtime. However, with the present invention that collects receiver type information, inlining of methods may be advantageously performed. Additionally, the methods that are inlined may be saved with receiver type information so that a subsequent execution of the program may take advantage of the inlining information and inline the same methods, if desired. If it is determined that the method should be recompiled, it is recompiled at step 755.

At some point, the collection of receiver type information will be stopped and saved as shown at a step 761. Step 761 is shown at the end of the flowchart but may occur at any point in the flowchart of Fig. 14. For example, during execution the collected receiver type information may be saved, either automatically or as a direct result of user input. Nevertheless, typically, the collection of receiver type information will stop at the normal termination of the program.

At step 755, the collected receiver type information was compiled into the method. Fig. 15 shows an embodiment of receiver type information being compiled into a method. A method 801 is shown with a polymorphic inline cache compiled into the computer code. The compiled code would typically be native machine instructions but the general flow of the polymorphic inline cache may be as shown. If object x is of class A, a direct call is made to the method to handle the message. Similarly, expressions are also present for classes B and C. If object x is of a class that has not been previously seen, it will be looked up by a call to the method lookup routine. This new receiver type information may be accumulated by the compiled method so that a later recompilation may take advantage of the new receiver type information (*see* step 759).

Saving the collected receiver type information allows a subsequent execution of the program to take advantage of the previously collected receiver type information. Thus, if a method is optimized, the method may quickly return to the optimized state in a subsequent execution of the program. Fig. 16 illustrates how the collected receiver type information may be stored in a Java class file.

A Java class file 851 includes multiple sections. A header section 853 includes a magic number (0xCAFEBAFE), minor version number and major version number. A constant pool 855 includes constants. An access information section 857 includes access flags and pointers to the this class and superclasses in the constant pool. Additionally, there is an interface section 859, fields section 861 and methods section 863. The layout of Java class files is described in more detail in The Java™ Virtual Machine, Tim Lindholm et al., Addison-Wesley, 1997, which is hereby incorporated by reference for all purposes.

At the end of class file 851 is an attributes section 865. Nested receiver types may be stored in the attributes section as shown. As shown there may be references to the call site where the receiver types were encountered. The references may be offsets within the method. The receiver types are nested in terms of how deeply in the methods the call site is located. Thus, if call sites are from an inlined method, these call sites will be nested deeper. In preferred embodiments, the nested receiver types are text that resemble the polymorphic inline cache that may be utilized to collect the receiver type information. A Java virtual machine implementation is required to silently ignore any or all attributes in attributes section 865 that it does not recognize. Accordingly, placing the collected receiver type information in this location will not adversely affect other virtual machine implementations.

Attributes section 865 shows nested receiver type information that includes inlining information. The numbers 15 indicate that the classes A, B and C were encountered at the call site referenced (*e.g.*, offset) by 15. For class A, the method invoked at this call site was inlined, which is indicated by the nested entry. The 4 indicates that at the call site referenced by 4 in the inlined method, a class R was encountered. Similarly, for class C, classes R and T were encountered at the call site referenced by 4 in the inlined method. Accordingly, the receiver type information may provide inlining information from a previous program execution.

As described above, an inline database is provided that stores collected receiver type information at call sites for a subsequent execution of the program. The next time the program is run, the Java virtual machine may utilize the previously collected receiver type information to optimize the program without taking the time to collect this information again. Accordingly, when a Java program reaches an optimized state and has the collected receiver type information stored in the class files, the program may operate more efficiently on subsequent runs. Additionally, since class files are machine independent, the efficiency is not limited to any one computer system or architecture.

Conclusion

While the above is a complete description of preferred embodiments of the invention, there are alternatives, modifications, and equivalents that may be used. It should be evident that the invention is equally applicable by making appropriate modifications to the
5 embodiments described above. For example, the inline database technique was described as utilizing a polymorphic inline cache to collect receiver type information. However, the invention is not limited to any one message dispatch technique and may be advantageously applied to other message dispatch techniques including those described herein. Therefore, the above description should not be taken as limiting the scope of the
10 invention that is defined by the meets and bounds of the impended claims along with their full scope of equivalents.

**RED-LINED
SUBSTITUTE SPECIFICATION**

0906075.062801
108230.52096860

PATENT APPLICATION

**INLINE DATABASE FOR RECEIVER TYPES IN OBJECT-
ORIENTED SYSTEMS**

Inventors: **LARS BAK**
3782 Corina Way
Palo Alto, California 94303
a Citizen of Denmark

URS HÖLZLE
7220 Davenport Road, #105
Goleta, California 93117
a Citizen of Switzerland

Assignee: **SUN MICROSYSTEMS, INC.**

~~2050 Gopala Avenue
Mountain View, California 94035
a Corporation of the State of California~~

Status: Large Entity

PATENT

5
**INLINE DATABASE FOR RECEIVER TYPES IN OBJECT-ORIENTED
SYSTEMS**

CROSS REFERENCE TO RELATED APPLICATIONS

10 This application is a continuation of U.S. Patent Application No. 08/944,332,
entitled "INLINE DATABASE FOR RECEIVER TYPES IN OBJECT-ORIENTED
SYSTEMS."

This application is related to U.S. Patent Application No. 08/944,331, entitled
"SITE SPECIFIC MESSAGE DISPATCH IN OBJECT-ORIENTED SYSTEMS"; U.S.
15 Patent Application No. 08/944,735, entitled "METHOD AND APPARATUS FOR
PERFORMING BYTE-CODE OPTIMIZATION DURING PAUSES," now U.S. Patent
5,970,249; U.S. Patent Application No. 08/944,335, entitled "MIXED EXECUTION
STACK AND EXCEPTION HANDLING," now U.S. Patent 6,009,517; U.S. Patent
Application No. 08/944,326, entitled "METHOD AND APPARATUS FOR
20 IMPLEMENTING MULTIPLE RETURN SITES"; U.S. Patent Application No.
08/944,334, entitled "METHOD AND APPARATUS FOR DYNAMICALLY
OPTIMIZING BYTE-CODED PROGRAMS," now U.S. Patent 5,995,754; U.S. Patent
Application No. 08/944,330, entitled "METHOD AND APPARATUS FOR
DYNAMICALLY OPTIMIZING COMPILED ACTIVATIONS," now U.S. Patent
25 5,933,635; all filed concurrently herewith.

This application is also related to U.S. Patent Application No. 08/884,856, entitled
"INTERPRETING FUNCTIONS UTILIZING A HYBRID OF VIRTUAL AND
NATIVE MACHINE INSTRUCTIONS," filed June 30, 1997, and U.S. Patent
Application No. 08/885,008, entitled "INTERPRETER GENERATION AND
30 IMPLEMENTATION UTILIZING INTERPRETER STATES AND REGISTER

CACHING," now U.S. Patent 6,021,273, filed June 30, 1997, which are all incorporated herein by reference for all purposes in their entirety.

BACKGROUND OF THE INVENTION

The present invention relates to ~~the~~ increasing the execution speed of object-oriented programs. More specifically, the invention relates to utilizing information collected during execution of an object-oriented program for subsequent executions.

The fundamental idea behind object-oriented languages is the combination of both data and the methods (or functions) that operate on that data into a single unit, which is called an object. An object's functions typically provide the only way to access the data

that is encapsulated by the object. The data is accessed by sending a message to the object instructing the object to invoke the method specified by the message.

Efficient message dispatch is of paramount importance in object-oriented languages. This is because message dispatch is a very frequent operation in object-oriented programs and is performed at runtime; therefore, it should be as fast as possible. Message dispatch, however, is far from being a trivial operation. Unlike procedural programming languages (e.g., the C programming language) that can determine a function's address before runtime, object-oriented languages must determine the method that handles a message that has been dispatched to a receiver object at runtime, and it may involve an extensive search.

In order to better understand the complexities of message dispatch, Fig. 1 shows a class hierarchy including methods of each class. A class hierarchy 1 includes at its root a parent class A 3 that defines two virtual functions foo() and bar(). Virtual functions are functions that may be defined in a parent class and redefined in the children classes. Class B 5 and class C 7 ~~also~~ inherit the data and methods of the parent class A. As shown, class B does not redefine either of the virtual functions foo and bar. However, class C redefines the virtual function foo. When an object of class C is requested to invoke the method foo, the method invoked will be the method defined by the class C, not the parent class A. Classes D 9 and E 11 also redefine the method foo.

As it is generally impossible to determine the class of an object statically, the search for the correct method is performed at runtime, during message dispatch. There are many known techniques for implementing method dispatch. For example, Fig. 2 shows an inline cache. Assume a method 51 was originally as follows:

```
main()
{
  ...
  x.foo();
  ...
}
```

Thus, the method main includes a statement x.foo() in order to invoke the method foo of object x.

Another technique for implementing message dispatch is the use of a polymorphic inline cache as shown in Fig. 3. As before, a method 101 originally included a method dispatch `x.foo()`. With a polymorphic inline cache, a stub 103 is generated such that it is able to perform the message dispatch for different receiver types. The original message dispatch is overwritten with a call to the polymorphic inline cache stub 103. Each time a new receiver type is encountered, a statement is added to the stub. As shown, three different receiver types ~~—~~have been encountered thus far. If the receiver type ~~—~~has been encountered, a call is made to the method to handle the message for that receiver type. Otherwise, the method lookup routine is called to determine the appropriate method to handle the message. Typically, a new statement will be added to stub 103 in order to handle each new receiver type.

The polymorphic inline cache is more flexible than the inline cache as it is able to handle multiple receiver types. However, a drawback of the polymorphic inline cache is that as more receiver types are encountered, the stub continues to grow and it becomes less and less efficient at performing message dispatch. For example, the system may need to plow through multiple if statements before finding the right method to handle the message.

Fig. 4 shows another message dispatch technique called hashing. In hashing, the original message dispatch in a method 151, `x.foo()`, is overwritten with a call to a hash function 153. The hash function hashes the receiver type and the message in order to form a hash key, which is typically an index into a hash table 155. The hash table includes an index 157, receiver types 159, messages 161, and methods 163. Once the hash function hashes to a row in ~~the~~ hash table 155, the receiver type and message are retrieved from columns 159 and 161 of the hash table. If the receiver type and message at the call site match the receiver type and message in the row of the hash table, the method specified in column 163 of hash table 155 is invoked. Otherwise, a method lookup routine is called in order to find the correct method. Typically, the new method is then added to the hash table.

Although hashing is the most flexible message dispatch technique we have described thus far, it is more computationally and storage intensive ~~---than~~ the other techniques. Another drawback of the message dispatch techniques we have described is that none of the techniques are site specific. In other words, none of the message dispatch techniques provide the flexibility of handling message dispatches at different call sites in a different manner. Other method dispatch techniques are described in "Message Dispatch on Pipelined Processors," by K. Driesen et al., ECOOP, 1995, which is hereby incorporated by reference for all purposes.

None of the message dispatch techniques described above allow for the receiver type information that is collected to be utilized during a subsequent execution of the program. Additionally, it would be desirable if inlining information was available during subsequent program executions. Accordingly, there is a need for techniques that allow for collected receiver type and/or inlining information to be utilized in later runs of the program.

SUMMARY OF THE INVENTION

Embodiments of the present invention provide innovative techniques for utilizing collected receiver type and/or inlining information during later program executions. By allowing a program to take advantage of information from previous executions, the program may become quickly optimized without requiring the information to be collected again. Therefore, an object oriented-program may become increasingly more efficient each time it is run until it reaches a steady optimized state. Several embodiments of the invention are described below.

In one embodiment, the invention provides a method of implementing message dispatch for an object-oriented program. Receiver type information is collected at a site of a method that dispatches messages to receiver objects. The collected receiver type information is then saved for a subsequent execution of the program. In preferred embodiments, the receiver type information is collected both during interpreted and compiled execution of the method.

In another embodiment, the invention provides a method of implementing message dispatch for an object-oriented program. During interpretation, collecting receiver type information at a site of a method that dispatches messages to receiver objects. The receiver type information may include each different receiver type and a reference to the site for each different receiver type to which messages were dispatched from the site. It is then determined that it would be desirable to compile the method that includes the site that dispatches messages to receiver objects. When the method is compiled, the receiver type information is included at the site in the compiled method. Also, the receiver type information is saved for a subsequent execution of the program. In some embodiments, the receiver type information is collected in a polymorphic inline cache.

In another embodiment, the invention provides a data structure stored by a computer readable medium. At least one receiver type is stored by the computer readable medium with nested receiver types. The nested receiver types are of receiver types that

were dispatched messages at message dispatch sites in a method. In preferred embodiments, the data structure is saved in an attributes section of a Java class file.

Other features and advantages of the invention will become readily apparent upon review of the following detailed description in association with the accompanying

5 drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates a class hierarchy of classes including virtual functions in an object-oriented environment.

Fig. 2 illustrates an inline cache technique for object-oriented message dispatch.

Fig. 3 illustrates a polymorphic inline cache technique for object-oriented message dispatch.

Fig. 4 illustrates a hashing technique for object-oriented message dispatch.

Fig. 5 illustrates an example of a computer system that may be utilized to execute the software of an embodiment of the invention.

Fig. 6 illustrates a system block diagram of the computer system of Fig. 5.

Fig. 7 illustrates how a Java source code program is executed.

Fig. 8 illustrates an embodiment of a site specific inline cache technique for object-oriented message dispatch.

Fig. 9 shows a high level flowchart of an embodiment of a site specific inline cache technique for object-oriented message dispatch.

Fig. 10 shows a high level flowchart of an embodiment of a site specific message dispatch technique.

Fig. 11 shows a flowchart of dynamically determining if another message dispatch technique is desirable at a specific site.

Fig. 12 illustrates an embodiment of a site specific message dispatch technique that utilizes NOP instructions to count the number of different receiver types that have been encountered.

Fig. 13 shows a high level flowchart of an inline database that may be utilized to save receiver type information for subsequent execution of a program.

Fig. 14 shows a flowchart of collecting receiver type information during interpretation and compiled execution of a program for an inline database.

Fig. 15 illustrates a method that has been compiled to include a polymorphic inline cache technique for object-oriented message dispatch.

Fig. 16 illustrates a Java class file in which receiver type information has been stored in the attributes section of the class file.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Definitions

Function – A software routine (also called a subroutine, procedure, member function, and method).

Message dispatch – A process of determining the method to handle a message that has been sent to the object into an object-oriented environment.

Receiver object (or receiver) – The object that has been sent a message in an object-oriented environment.

Receiver type – The class to which a receiver object belongs.

Call site (or site) – The location or area in a program where a message is dispatched to an object.

Overview

In the description that follows, the present invention will be described in reference to preferred embodiments that are designed for a Java™ virtual machine. In particular, examples will be described that are directed to IBM personal computers. However, the invention is not limited to any particular language, computer architecture, or specific implementation. Therefore, the description of the embodiments that follows is for purposes of illustration and not limitation.

Fig. 5 illustrates an example of a computer system that may be used to execute the software of an embodiment of the invention. Fig. 5 shows a computer system 301 that includes a display 303, screen 305, cabinet 307, keyboard 309, and mouse 311. Mouse 311 may have one or more buttons for interacting with a graphical user interface. Cabinet 307 houses a CD-ROM drive 313, system memory and a hard drive (*see* Fig. 6) which may be utilized to store and retrieve software programs incorporating computer code that implements the invention, data for use with the invention, and the like. Although CD-ROM 315 is shown as an exemplary computer readable storage medium, other computer

readable storage media including floppy disk, tape, flash memory, system memory, and hard drive may be utilized. Additionally, a data signal embodied in a carrier wave (e.g., in a network including the Internet) may be the computer readable storage medium.

Fig. 6 shows a system block diagram of computer system 301 used to execute the software of an embodiment of the invention. As in Fig. 5, computer system 301 includes monitor 303 and keyboard 309, and mouse 311. Computer system 301 further includes subsystems such as a central processor 351, system memory 353, fixed storage 355 (e.g., hard drive), removable storage —357 (e.g., CD-ROM drive), display adapter 359, sound card 361, speakers 363, and network interface 365. Other computer systems suitable for use with the invention may include additional or fewer subsystems. For example, another computer system could include more than one processor 351 (i.e., a multi-processor system) or a cache memory.

The system bus architecture of computer system 301 is represented by arrows 367. However, these arrows are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be utilized to connect the central processor to the system memory and display adapter. Computer system 301 shown in Fig. 6 is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

The Java™ programming language was developed by Sun Microsystems. Typically, computer programs written in the Java programming language are compiled into bytecodes or Java virtual machine instructions that are then executed by a Java virtual machine. The bytecodes are stored in class files that are input into the Java virtual machine for interpretation. Fig. 7 shows a progression of a simple piece of Java source code through execution by an interpreter, the Java virtual machine.

Java source code 401 includes the classic Hello World program written in Java. The source code is then input into a bytecode compiler 403 that compiles the source code into bytecodes. The bytecodes are virtual machine instructions as they will be executed by a software emulated computer. Typically, virtual machine instructions are generic (i.e., not designed for any specific microprocessor or computer architecture) but this is not

required. The bytecode compiler outputs a Java class file 405 that includes the bytecodes for the Java program.

The Java class file is input into a Java virtual machine 407. The Java virtual machine is an interpreter that decodes and executes the bytecodes in the Java class file.

5 The Java virtual machine is an interpreter, but is commonly referred to as a virtual machine as it emulates a microprocessor or computer architecture in software (e.g., the microprocessor or computer architecture may not exist in hardware). In order to increase execution speed of the Java virtual machine, methods may be compiled into native machine instructions for faster execution. In the description of preferred embodiments
10 that follows, compilation will be used to refer to the translation of Java virtual machine instructions into native machine instructions.

Site Specific Inline Cache

The invention provides a site specific inline cache technique for implementing
15 message dispatch in object-oriented environments. Receiver type information is stored at the call site, which allows each call site to vary according to runtime characteristics of the call site. This allows message dispatch to not only be more efficient, but also to be more flexible.

Fig. 8 illustrates a site specific inline ~~cache~~ technique according to one
20 embodiment of the invention. As described in the above ———Background of the Invention, a method 51 originally included a message x.foo() for dispatch. The first time that this message is dispatched, the system may call a method lookup routine to determine the location of the appropriate method to handle the message. Alternatively, the system may perform a static analysis prior to runtime to determine what is likely to be the correct
25 method.

With a site specific inline cache, once the class of the object that should handle the message is determined, an instruction is generated at the call site to save the receiver type or class. The message dispatch is then overwritten with a call to the specific method that

should handle the message. Thus, a message dispatch becomes two instructions, one that saves a receiver type and a call to the appropriate method.

As shown, method 451 includes a move instruction that stores an —ID for the class A into a special register. The register is called a special register only for identification purposes. What is important is that if a register is utilized, the register should be otherwise unused at this call site. Although utilizing registers may provide the most efficient execution speed, other storage mechanisms may be utilized. For example, the receiver type may be stored in system memory or sent as a parameter to the message handling method.

Once the receiver type is saved, a call is made to a method 453 that handles the message, which is shown as A::foo(). A prolog 455 checks if the object x is of the saved receiver type, which is retrieved from the special register. In the example shown, class A is the stored receiver type. Thus, if object x is of class A, a jump is made to a method code 457 of A::foo(). Otherwise, a method lookup routine is called to locate the appropriate method to handle the message. Then, the call site may be updated to save the receiver type of object x and a call to the method found by the method lookup routine.

The receiver type stored at the call site is therefore a predicted receiver type. It is the receiver type that the system predicts will be the receiver type of the next object to receive the message. If the predicted receiver type matches the receiver type of the object, the message dispatch is more efficient. Additionally, each message dispatch site may have receiver type information stored that is particular to that call site and another site will not overwrite the stored receiver type information.

Fig. 9 shows a flowchart of an embodiment of the site specific inline cache technique for message dispatch. At a step 501, a predicted receiver type is saved at a site where a message is dispatched to a receiver object. In preferred embodiments, the receiver type is saved by utilizing a move instruction to place the receiver type in a register. However, other methods of saving the receiver type may be utilized that are well known to those of skill in the art, including the use of system memory or parameter passing.

At a step 503, the message is sent to the receiver object. The message is sent by a direct call to (or invocation of) the method that handles the message instead of utilizing a method lookup routine.

Once the receiver object receives the message, it is verified that the receiver object is of the saved predicted receiver type at a step 505. In preferred embodiments, this verification is performed in the prolog of the method (see Fig. 8). If the receiver object is of the same type as the saved predicted receiver type at a step 507, the method that handles the message is executed at a step 509. Step 509 may include performing a jump to the start of the method code that handles the message.

If the receiver object is not of the same type as the saved predicted receiver type, the predicted receiver type at the call site is changed to the receiver type of the receiver object at a step 511. The method lookup routine is called to obtain a new method at a step 513. The method lookup routine utilized may be any that are known to those of skill in the art, including dispatch table searches and virtual function tables. The method lookup routine provides the appropriate method to handle the message. The message dispatch at the call site is then changed to the method returned by the method lookup routine at a step 515.

As an example, assume that in Fig. 8, object x is of class A, so the predicted receiver type stored at the call site in method 451 is class A. However, if object x is subsequently of class B, prolog 455 may modify the move instruction to store the class B in the special register as the predicted receiver type. Additionally, the message dispatch will be modified to invoke the method returned by the method lookup routine. Subsequent message dispatches at this call site that are directed at a receiver object of class B will then be more efficiently dispatched.

In Fig. 9, the method is shown as being executed after step 515 that changes the method at the site. The step of executing the method may actually occur within step 513 or at other locations. Therefore, in the flowcharts shown, steps may be reordered, added and deleted according to the specific implementation without departing from the spirit of the invention.

As described above, a site specific inline cache technique is provided that saves a predicted receiver type at a call site. By storing a predicted receiver type at a call site, message dispatch may be more efficient and flexible. For example, the operations at each call site may be insulated from each other so that one call site will not change the settings for another call site.

Site Specific Message Dispatch

The invention provides a site specific message dispatch technique that is flexible in allowing for different message dispatch techniques to be utilized at each site.

Additionally, the message dispatch technique utilized at a call site may change over time, during runtime. Once it is determined that a new or different message dispatch technique is desirable at a specific site, the new message dispatch technique is set up for use at the specific call site. Thus, message dispatch becomes more flexible not only in terms of call site, but also in terms of the message dispatch technique that is utilized at call during different times during program execution.

Fig. 10 shows a high level flowchart of an embodiment of site specific message dispatch. At a step 511, a first message dispatch technique is performed at a specific message dispatch site. The first message dispatch technique may be any number of message dispatch techniques known in the art. Preferably, the first message dispatch technique is a technique according to the present invention. For example, the first message technique may be an embodiment of the site specific inline cache described above. The first message dispatch technique may be determined statically before runtime or dynamically during runtime.

At a step 553, the system dynamically during runtime determines if a second message dispatch technique is desirable at the specific site. Any number of ways may be utilized to determine that a second message dispatch technique is desirable. For example, once the number of different receiver types encountered at this call site crosses a threshold, it may be desirable to switch to a different message dispatch technique. Additionally, a calculation may be performed on the frequency that the method lookup

routine is called so that if the frequency is too high, a new message dispatch may be warranted.

At any rate, once it is determined that a second message dispatch technique is desirable, the second message dispatch technique is performed at the specific site at a step

5 555. The mechanism for switching to a new message dispatch technique typically depends on the new technique itself. Now that the high level flow has been described, it may be beneficial to discuss in more detail an embodiment of the step of dynamically determining if a second message dispatch technique is desirable.

Fig. 11 shows an embodiment of dynamically determining if a second message
10 dispatch technique is desirable. The labels of first and second message dispatch techniques are presented as generic indications that one message dispatch technique may be switched to another at a specific call site. However, the invention is not limited to only two techniques, and may be advantageously applied to three or more techniques. Therefore, the labels of first and second are not an indication that the invention is limited
15 to only two message dispatch techniques.

The embodiment shown in Fig. 11 relies on the number of different receiver types encountered at a specific call site as an indicator of when the message dispatch should be changed. As an example, a site specific inline cache technique may be utilized until five different receiver types have been encountered at this call site. When a different receiver
20 type is utilized in this context, it means that there have been five instances where the predicted receiver type has been different than the actual receiver type. This would mean that approximately five times, the method lookup routine has been called and the predicted receiver type that is stored at the call site has changed. As an example, a call site that alternates five times between receiver objects of classes A and B may be counted
25 as five different receiver types.

Although a specific receiver type counter may be utilized at each call site, this may entail the use of memory and computer code to maintain the counter. In preferred embodiments, the counter is implemented as multiple no-operation (NOP) instructions preceding the method that handles the message. As will be described in more detail in

reference to Fig. 12, the number of NOP instructions that are implemented before the method indicates the number of different receiver types that have been encountered.

Referring to Fig. 11, the flowchart assumes that the method lookup routine has been called, otherwise, most likely, the current message dispatch technique is performing efficiently. At a step 601, the system dispatches messages to NOP instructions preceding the method that handles the message. The system counts the number of different receiver types by determining how many NOP instructions are executed before the method. Typically, this count will be determined by subtracting the address to where the message was dispatched for this site from the address of the actual method, including any prolog.

At a step 605, the count is compared to a predetermined number. The predetermined number is the number of different receiver types that may be encountered at a call site before it is desirable to switch to a different message dispatch technique. The predetermined number may be statically defined or calculated dynamically during runtime.

If the count is greater than the predetermined number at a step 607, a switch is made to the second message dispatch technique at a step 609. If the count is less than or equal to the predetermined number, this indicates that, at present, it is not desirable to switch to the second message dispatch technique. An example may help in illustrating how NOP instructions may be utilized as a counter.

Fig. 12 illustrates how NOP instructions may be utilized as a counter of the number of different receiver types that have been encountered at a specific call site. A method 651 is shown with a site specific inline cache as described previously. A method 653 handles the message, however, the method is preceded by a NOP section 655 that has multiple NOP instructions. NOP instructions may be chosen as they do nothing and may not unduly ~~effect~~ affect the operating speed of the method. Other instructions may be utilized instead of NOP instructions in other embodiments.

NOP section 655 precedes a prolog 657 and method code 659 of method 653. The NOP section is composed of a predetermined number of NOP instructions, typically the same number as the number of different receiver types that may be encountered at a call

site before a switch to a different message dispatch technique is initiated. Each time the method lookup routine is called in prolog 657, the move instruction is patched with the receiver type of object x. Additionally, the subsequent message dispatch (e.g., A::foo()) is changed to one of the NOP instructions in NOP section 655 of the new method found by the method lookup routine.

As shown, the message dispatch points to the third NOP instruction before method 653. If prolog 657 determines that object x is not of class A, the prolog calculates the difference between the address specific at the call site and the beginning of method 653, which is obtainable during runtime by a number of known methods. The difference or count indicates the number of different receiver types that have been encountered at this specific call site. If the count is greater than a predetermined number, prolog 657 may switch the message dispatch technique utilized at this call site.

Utilizing NOP instructions before a method is easy to implement, fast, and does not require an excessive amount of computer code to keep the counter. Additionally, the NOP instructions in NOP section 655 may be referenced by multiple different call sites without any impact on each other.

The first message dispatch technique was shown as being a site specific inline cache. The second message dispatch technique may be a polymorphic inline cache or hashing technique, as each are designed to handle multiple receiver types. Of course, there is no limitation on the message dispatch techniques that may be utilized with the invention. The specific message dispatch techniques have been described herein to aid the reader's understanding.

As described above, a site specific message dispatch technique is provided that allows for a new message dispatch technique to be utilized at a specific call site when it is deemed desirable. By allowing each call site to switch message dispatch techniques when it becomes desirable, message dispatch may be more efficient and flexible. Additionally, the operations at each call site may be insulated from each other so that one call site will not change the settings for another call site.

Inline Database

The invention provides a technique of saving receiver type and/or inlining information for a subsequent execution of a program. Receiver type and inlining information may be collected during the runtime of the program, the collected runtime information may then be stored so that a subsequent execution of the program can realize the benefits of the collected information without having to collect the information again. This allows the program to achieve an optimized state during runtime and quickly return to that optimized state in a subsequent execution of the program.

Fig. 13 shows a high level flowchart of an embodiment of the invention that collects receiver type information and stores it for a subsequent execution of a program. At a step 701, the system collects receiver type information at a message dispatch site of a method of a program. The receiver type information may be collected by any of the message dispatch techniques described here or otherwise known in the art. The receiver type information is preferably specific to a call site so receiver type information may be collected for multiple call sites.

Once the receiver type information for a specific call site is collected, it is saved for a subsequent execution of the program at a step 703. The receiver type information may be saved at the normal termination of the program or at some point during runtime, either with or without a user's input. It may be beneficial to describe in more detail an embodiment of collecting receiver type information.

Fig. 14 shows a flowchart of an embodiment of collecting receiver type information. At a step 751, receiver type information is collected during interpretation of the method. Typically, methods are interpreted in a Java virtual machine until it is determined that it would be beneficial to compile the method. In systems where methods are immediately compiled, the flow may begin at a subsequent step.

As receiver type information is collected during interpretation, such as the different receiver types that were encountered at specific call sites, one or more than one message dispatch technique may be utilized. At some point, there may be a question of whether this method should be compiled at a step 753. For example, if a method has been

interpreted more than a predetermined number of times, it may be determined that the method should be compiled. Otherwise, receiver type information may be continually collected during interpretation of the method.

At a step 755, the method is compiled and the collected receiver type information is compiled into the method. This way, the compiled method will realize the benefits of the collected type information. Additionally, in preferred embodiments, computer code is generated in the compiled method to continue collecting type information at a step 757.

In preferred embodiments, the system also maintains a working set of compiled methods. This working set may also be saved off with the collected type information so that in a subsequent execution of the program, the system may compile methods from the working set when the system has available processing capabilities (*e.g.*, when the system is waiting on user input).

Once a method is compiled, it may be beneficial to recompile the method at some point in time. At a step 759, it is determined if the method should be recompiled. As an example, it may be beneficial to recompile the method if a substantial amount of new receiver type information has been collected. In preferred embodiments, a method may be recompiled if it is determined (*e.g.*, heuristics) that a method should be inlined into this method. Inlining means that instead of calling the method, the method is compiled into the calling method. For example, if it is determined that it would be beneficial to inline a method, the method is compiled into the calling method and thus the calling method now includes the call sites of the inlined method.

It is difficult, if ~~not~~ impossible for conventional object-oriented systems to inline methods since the receiver object may not be determined till runtime. However, with the present invention that collects receiver type information, inlining of methods may be advantageously performed. Additionally, the methods that are inlined may be saved with ~~the~~ receiver type information so that a subsequent execution of the program may take advantage of the inlining information and inline the same methods, if desired. If it is determined that the method should be recompiled, it is recompiled at step 755.

At some point, the collection of receiver type information will be stopped and saved as shown at a step 761. Step 761 is shown at the end of the flowchart but may occur at any point in the flowchart of Fig. 14. For example, during execution the collected receiver type information may be saved, either automatically or as a direct result of user input. Nevertheless, typically, the collection of receiver type information will stop at the normal termination of the program.

At step 755, the collected receiver type information was compiled into the method. Fig. 15 shows an embodiment of receiver type information being compiled into a method. A method 801 is shown with a polymorphic inline cache compiled into the computer code. The compiled code would typically be native machine instructions but the general flow of the polymorphic inline cache may be as shown. If object x is of class A, a direct call is made to the method to handle the message. Similarly, expressions are also present for classes B and C. If object x is of a class that has not been previously seen, it will be looked up by a call to the method lookup routine. This new receiver type information may be accumulated by the compiled method so that a later recompilation may take advantage of the new receiver type information (*see* step 759).

Saving the collected receiver type information allows a subsequent execution of the program to take advantage of the previously collected receiver type information. Thus, if a method is optimized, the method may quickly return to the optimized state in a subsequent execution of the program. Fig. 16 illustrates how the collected receiver type information may be stored in a Java class file.

A Java class file 851 includes multiple sections. A header section 853 includes a magic number (0xCAFEBAFE), minor version number and major version number. A constant pool 855 includes constants. An access information section 857 includes access flags and pointers to ~~this~~ this class and superclasses in the constant pool. Additionally, there is an interface section 859, fields section 861 and methods section 863. The layout of Java class files is described in more detail in The Java™ Virtual Machine, Tim Lindholm et al., Addison-Wesley, 1997, which is hereby incorporated by reference for all purposes.

00000075.002001

At the end of class file 851 is an attributes section 865. Nested receiver types may be stored in the attributes section as shown. As shown, there may be references to the call site where the receiver types were encountered. The references may be offsets within the method. The receiver types are nested in terms of how deeply in the methods the call site is located. Thus, if call sites are from an inlined method, these call sites will be nested deeper. In preferred embodiments, the nested receiver types are text that resemble the polymorphic inline cache that may be utilized to collect the receiver type information. A Java virtual machine implementation is required to silently ignore any or all attributes in attributes section 865 that it does not recognize. Accordingly, placing the collected receiver type information in this location will not adversely affect other virtual machine implementations.

Attributes section 865 shows nested receiver type information that includes inlining information. The numbers 15 indicate that the classes A, B and C were encountered at the call site referenced (*e.g.*, offset) by 15. For class A, the method invoked at this call site was inlined, which is indicated by the nested entry. The 4 indicates that at the call site referenced by 4 in the inlined method, a class R was encountered. Similarly, for class C, classes R and T were encountered at the call site referenced by 4 in the inlined method. Accordingly, the receiver type information may provide inlining information from a previous program execution.

As described above, an inline database is provided that stores collected receiver type information at call sites for a subsequent execution of the program. The next time the program is run, the Java virtual machine may utilize the previously collected receiver type information to optimize the program without taking the time to collect this information again. Accordingly, when a Java program reaches an optimized state and has the collected receiver type information stored in the class files, the program may operate more efficiently on subsequent runs. Additionally, since class files are machine independent, the efficiency is not limited to any one computer system or architecture.

Conclusion

While the above is a complete description of preferred embodiments of the invention, there ~~are~~ are alternatives, modifications, and equivalents that may be used. It should be evident that the invention is equally applicable by making appropriate

5 modifications to the embodiments described above. For example, the inline database technique was described as utilizing a polymorphic inline cache to collect receiver type information. However, the invention is not limited to any one message dispatch technique and may be advantageously applied to other message dispatch techniques including those described herein. Therefore, the above description should not be taken as limiting the
10 scope of the invention that is defined by the meets and bounds of the impended claims along with their full scope of equivalents.